Generalized Search Trees for Database Systems

Matthias Springer

Hasso-Plattner-Institut

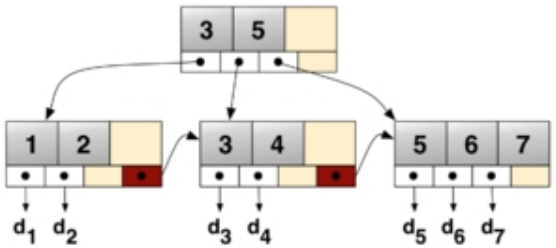
11. Juni 2012

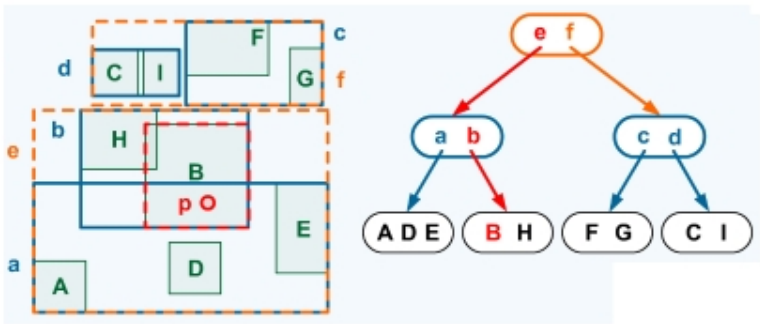
Gliederung

1. **Motivation**
2. Struktur eines GiST
3. Suche
4. Einfügen
5. Effizienz
6. Zusammenfassung

Daten- und Anfragetypen

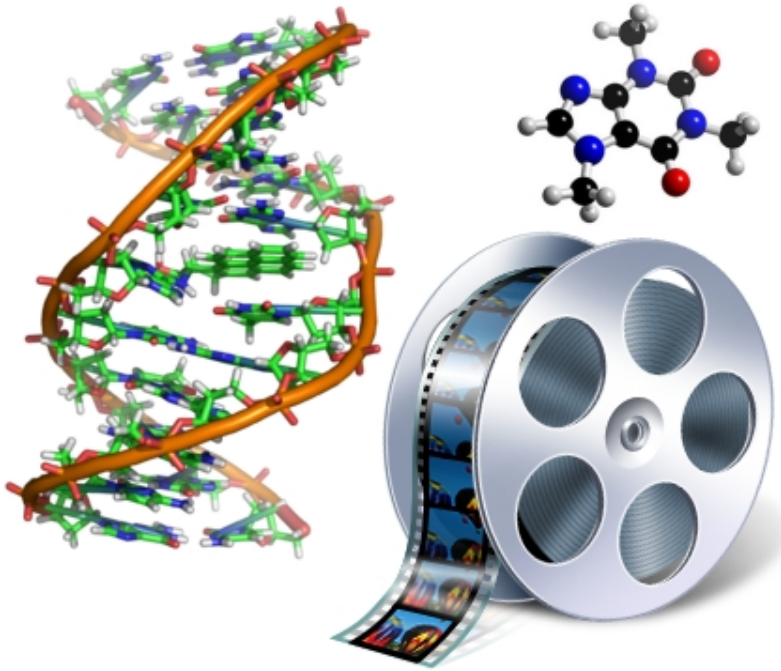
B^+- und R-Bäume





- Daten mit linearer Ordnung, räumliche Daten
- Bereichsanfragen, etc.

Neue Daten- und Anfragetypen



- Daten möglicherweise ohne lineare Ordnung
- Komplexe Anfragen

Die Autoren

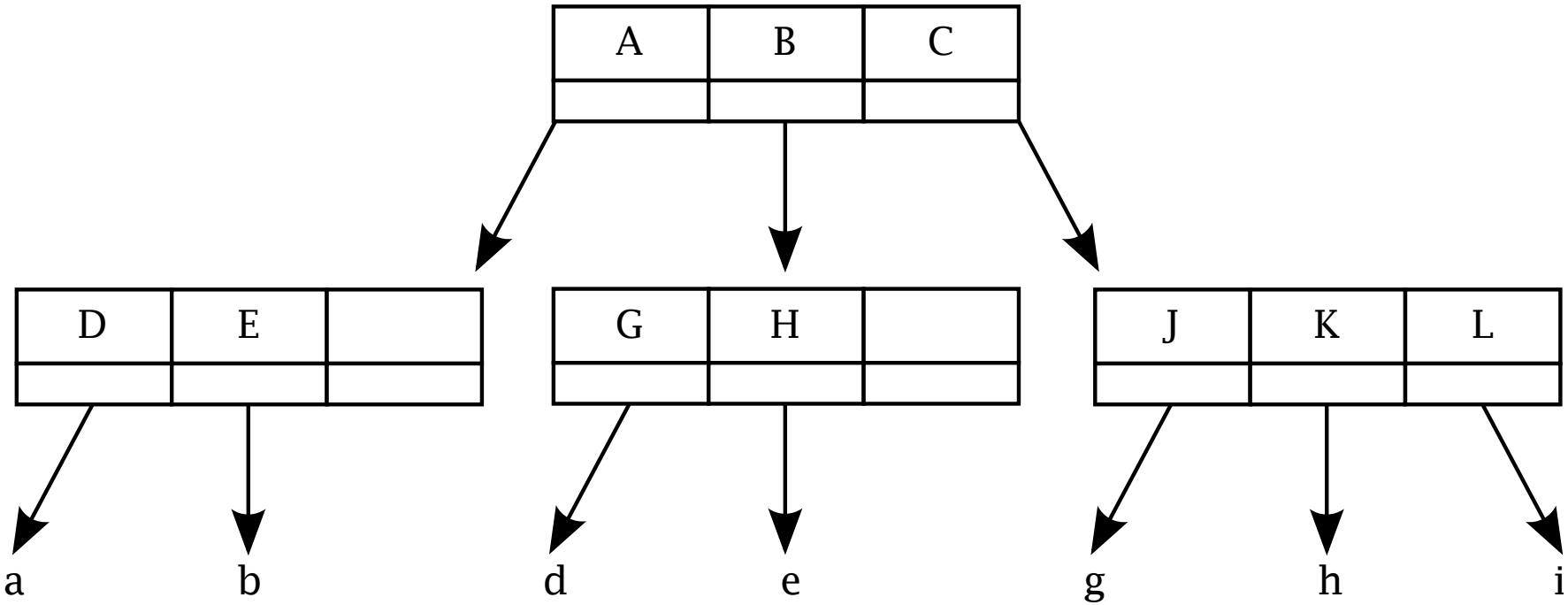


- **Joseph M. Hellerstein**: Professor an der University of California
- **Jeffrey Naughton**: Professor an der University of Wisconsin-Madison
- **Avi Pfeffer**: Associate Professor an der Harvard School of Engineering

Gliederung

1. Motivation
2. **Struktur eines GiST**
3. Suche
4. Einfügen
5. Effizienz
6. Zusammenfassung

Struktur eines GiST

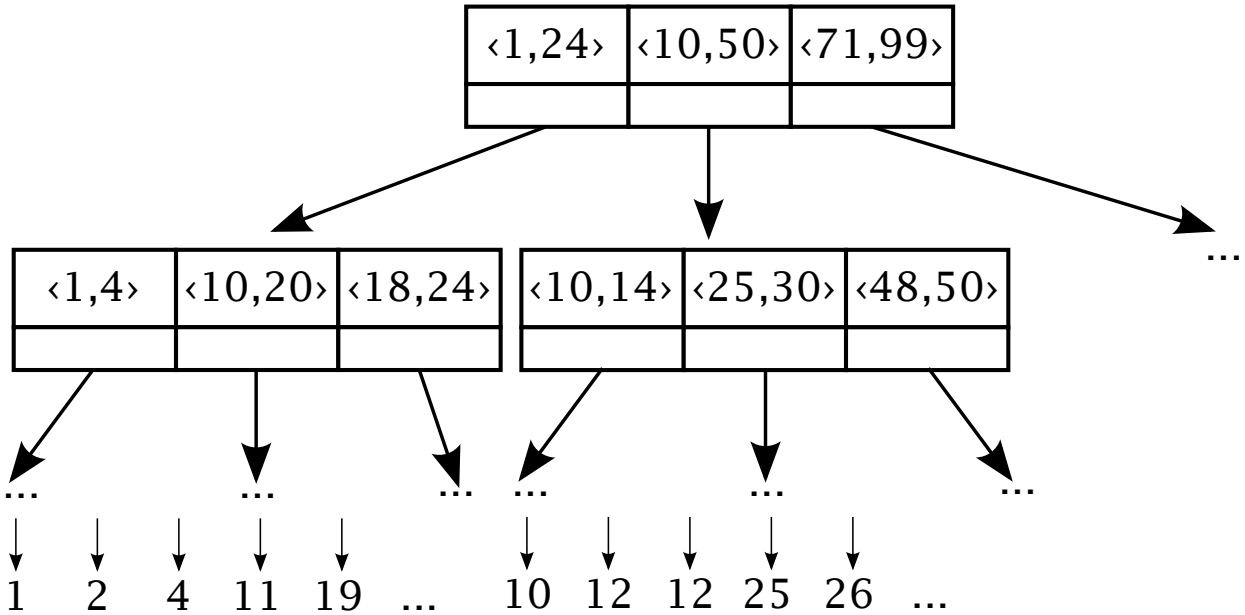


- Ballancierter Suchbaum
- Innere Knoten: Prädikat (Key), Zeiger Paare
- Key ist auch für Kindknoten erfüllt
- **Beispiel:** ist_student(t) $\wedge$ belegt_biob(t)

Prädikatenlogische Formel

- **Beispiel:** ist_student(t) $\wedge$ belegt_biob(t)
- ist_student : *Tupel* $\rightarrow \{T, \perp\}$
- Keys und Queries sind prädikatenlogische Formeln

Beispiel: Index für numerische Daten



- Query-Prädikate:
  - `range(`*from*`,` *to*`,` *t*`)`: Bereichsanfrage $[from, to]$
  - `equal(`*value*`,` *t*`)`: Gleichheit
  - `multiple(`*value*`,` *t*`)`: Alle Vielfachen von *value*
- Key-Prädikat: `contains(`*from*`,` *to*`,` *t*`)`, kurz $\langle from, to \rangle$

Methoden eines GiST

Pro GiST

- `Search(`R`, `q`)`
- `Insert(`E`)`
- `ChooseSubtree(`R`, `E`)`
- `Split(`L`, `E`)`
- `AdjustKeys(`L`)`

Pro Datentyp

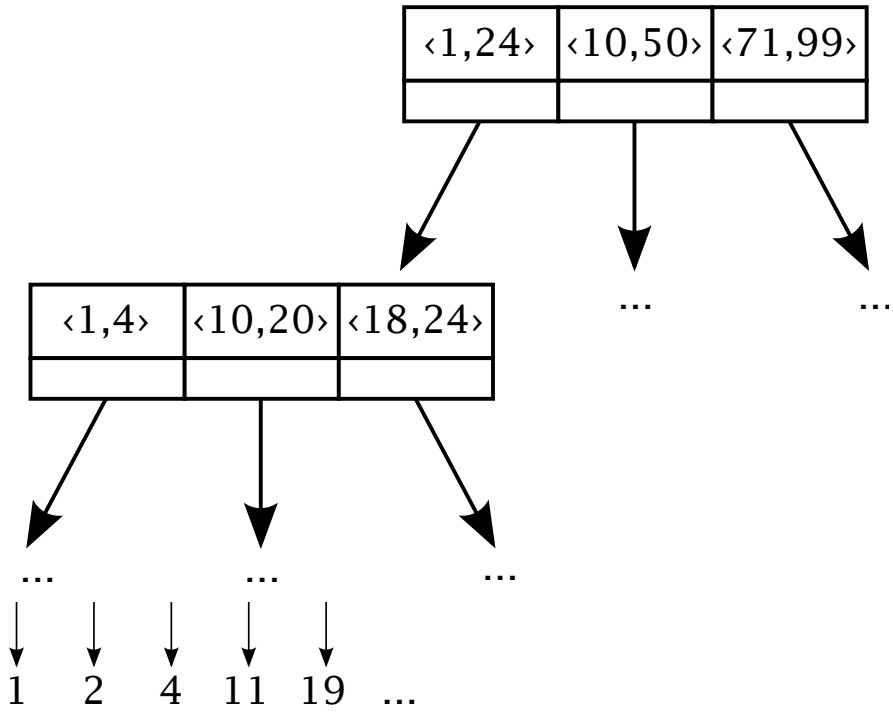
- Implementierung jedes Query-Prädikats
- `Consistent(`E`, `q`)`
- `Union(`P`)`
- `Penalty(`E_1`, `E_2`)`
- `PickSplit(`P`)`
- `Compress(`E`)`
- `Decompress(`E`)`

Gliederung

1. Motivation
2. Struktur eines GiST
3. **Suche**
4. Einfügen
5. Effizienz
6. Zusammenfassung

`Search(`R`, `q`)`

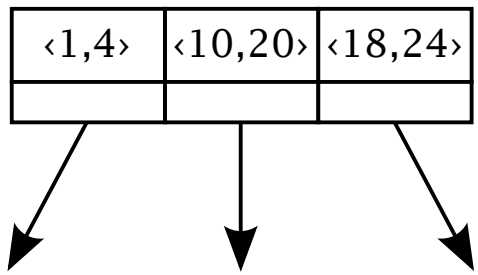
- **Eingabe:**
  - R: Startknoten
  - q: Query-Prädikat
- **Ausgabe:** Ein/mehrere Blätter
- **Algorithmus:**
  - Innere Knoten: Rekursiver Aufruf für alle Keys p mit `Consistent(`p`, `q`)`
  - Blätter: Prüfe Prädikat exakt und gib Element ggf. aus



`Consistent(`P`, `q`)`

- **Eingabe:**
  - P: Einzelner Key (Prädikat)
  - q: Query-Prädikat
- **Ausgabe:**
  - `false`, falls $P \wedge q$ sicher unerfüllbar ist, `true` sonst

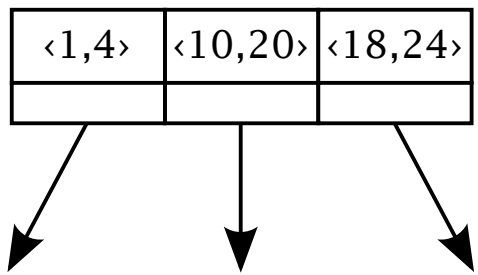
- **Beispiele:**
  - $\texttt{Consistent}(\langle 1, 4\rangle,\ \texttt{equal}(3)) = T$
  - $\texttt{Consistent}(\langle 1, 4\rangle,\ \texttt{equal}(5)) = \bot$
  - $\texttt{Consistent}(\langle 1, 4\rangle,\ \texttt{range}(3, 9) = T$
  - $\texttt{Consistent}(\langle 1, 4\rangle,\ \texttt{range}(5, 9) = \bot$
  - $\texttt{Consistent}(\langle 1, 4\rangle,\ \texttt{multiple}(10)) = \bot$
  - $\texttt{Consistent}(\langle 18, 24\rangle,\ \texttt{multiple}(10)) = T$



`Consistent(`P`, `q`)`

- **Eingabe:**
  - P: Einzelner Key (Prädikat)
  - q: Query-Prädikat
- **Ausgabe:**
  - Heuristik für Erfüllbarkeit
  - `false`, falls $P \wedge q$ sicher unerfüllbar ist, `true` sonst

- **Beispiele:**
  - $\texttt{Consistent}(\langle 1,4\rangle,\ \texttt{equal}(3)) \in \{T\}$
  - $\texttt{Consistent}(\langle 1,4\rangle,\ \texttt{equal}(5)) \in \{\bot, T\}$
  - $\texttt{Consistent}(\langle 1,4\rangle,\ \texttt{range}(3,9) \in \{T\}$
  - $\texttt{Consistent}(\langle 1,4\rangle,\ \texttt{range}(5,9) \in \{\bot, T\}$
  - $\texttt{Consistent}(\langle 1,4\rangle,\ \texttt{multiple}(10)) \in \{\bot, T\}$
  - $\texttt{Consistent}(\langle 18,24\rangle,\ \texttt{multiple}(10)) \in \{T\}$



Gliederung

1. Motivation
2. Struktur eines GiST
3. Suche
4. **Einfügen**
5. Effizienz
6. Zusammenfassung

`Insert(`E`)`

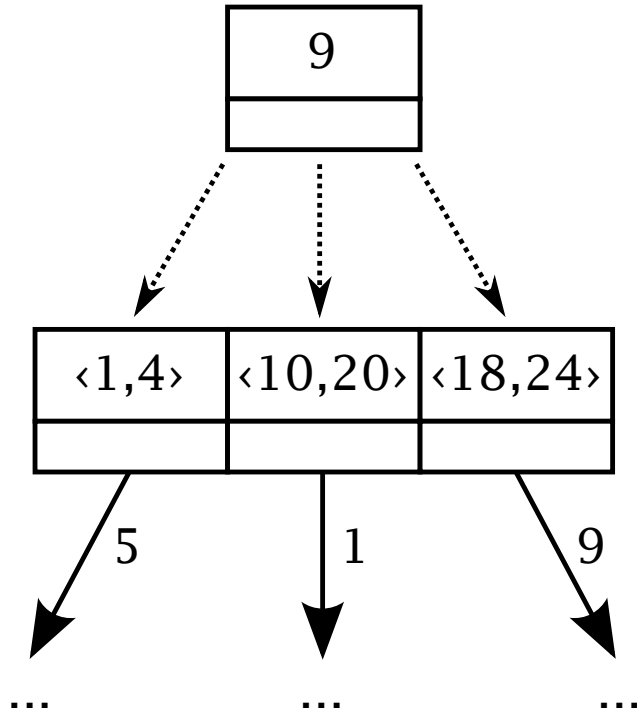
- **Eingabe:** $E = (p, \texttt{ptr})$: einzufügendes Key-Pointer-Paar
- **Danach:** GiST mit zusätzlichem Element E
- **Algorithmus:**
  1. Suche Knoten $L =$ `ChooseSubtree(`R`, `E`)` zum Einfügen
  2. Falls zu wenig Platz: Aufteilen mit `Split(`L`, `E`)`
  3. Füge E in L ein
  4. Keys anpassen mit `AdjustKeys(`L`)`

`ChooseSubtree(`R`, `E`)`

- **Eingabe:**
  - R: Wurzel des Teilbaums, in den eingefügt wird
  - $E = (p, \texttt{ptr})$: einzufügendes Key-Pointer-Paar
- **Ausgabe:** Knoten, in den E eingefügt wird
- **Algorithmus:**
  - Innere Knoten:
    - Finde Eintrag F mit kleinster `Penalty(`F`, `E`)`
    - Gib `ChooseSubtree(`F`.ptr, `E`)` zurück
  - Unterste Ebene: Gib R zurück

Penalty(F, E)

- **Eingabe:**
  - F: Key für einen Knoten
  - E: Einzufügendes Tupel
- **Ausgabe:** Maß für das Wachstum von F
- Regel: Keys möglichst kompakt halten
- $\text{Penalty}(F, x) = \max\{F.left - x, 0\} + \max\{x - F.right, 0\}$



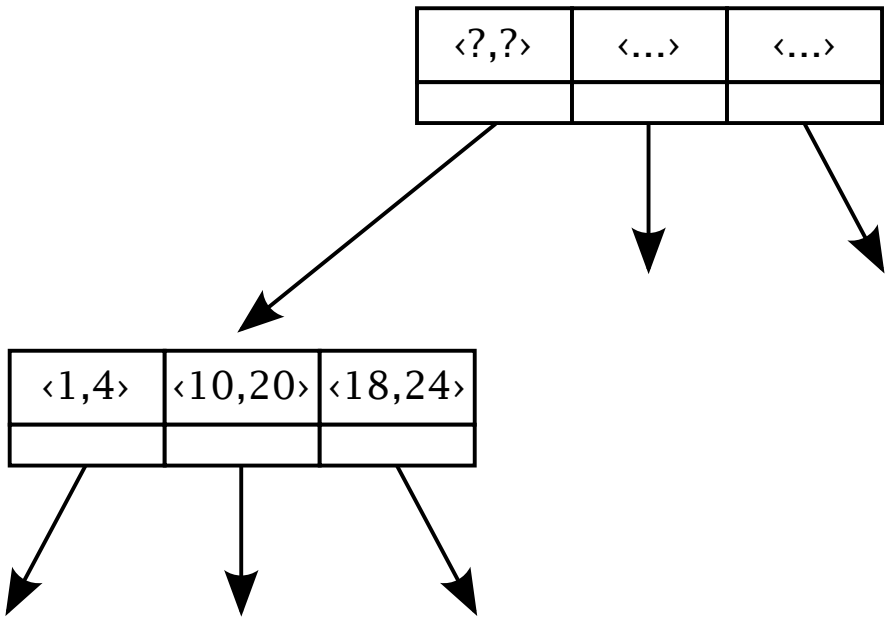
Split(L)

- **Eingabe:** L: Knoten, in den E eingefügt wird
- **Danach:** In L ist genügend Platz
- **Algorithmus:**
  1. Teile L in zwei gleich große Knoten L und L_2 auf (PickSplit)
  2. Füge (Union(L_2), $\texttt{ptr}_{L_2}$) zu parent(L) hinzu
  3. Passe Key von L in parent(L) mittels Union an

  - Falls kein Platz: Rufe Split rekursiv auf
  - Wurzel aufteilen: erzeuge neue Wurzel

Union(P)

- **Eingabe:** P: Menge an Key-Pointer-Paaren $(p_i, \texttt{ptr}_i)$
- **Ausgabe:** Prädikat a, sodass $\bigvee_i p_i \rightarrow a$

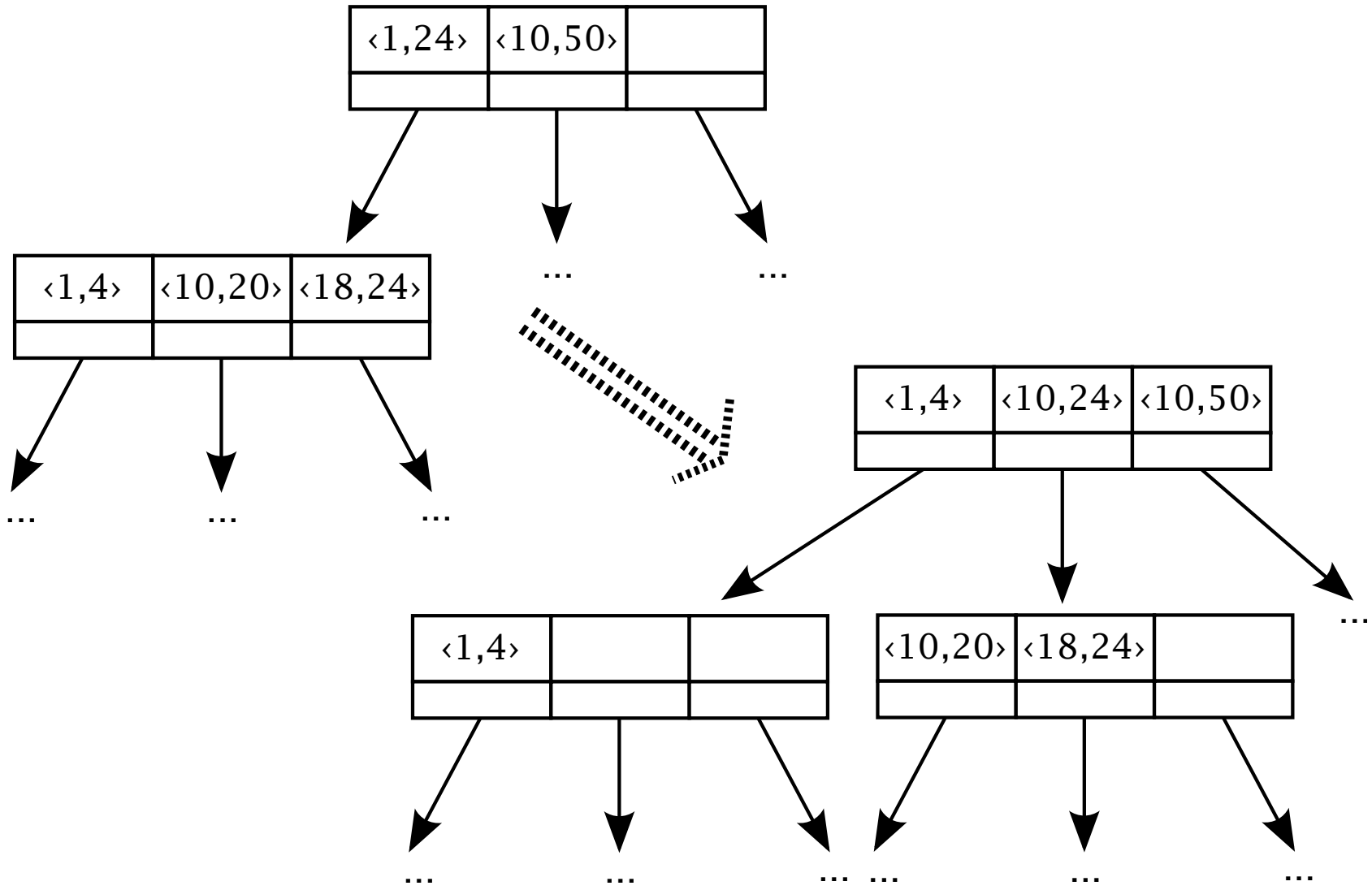


- **Beispiel:**

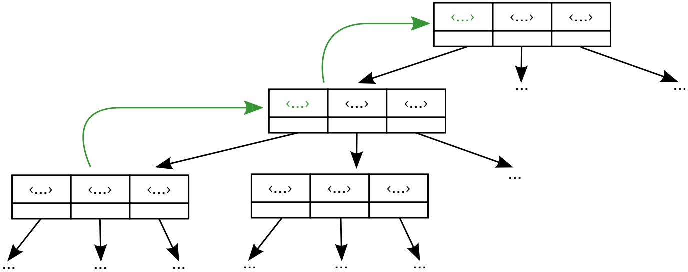
$$P = \left\{ \begin{array}{c} (\langle 1,4 \rangle, \texttt{ptr}_1) \\ (\langle 10,20 \rangle, \texttt{ptr}_2) \\ (\langle 18,24 \rangle, \texttt{ptr}_3) \end{array} \right\}$$

  - $\langle 1,4 \rangle \vee \langle 10,20 \rangle \vee \langle 18,24 \rangle \rightarrow a$
  - $a \equiv \langle 1,24 \rangle$
  - $a \equiv \langle 0,100 \rangle$

Beispiel: `Split(`L`)`



AdjustKeys(L)



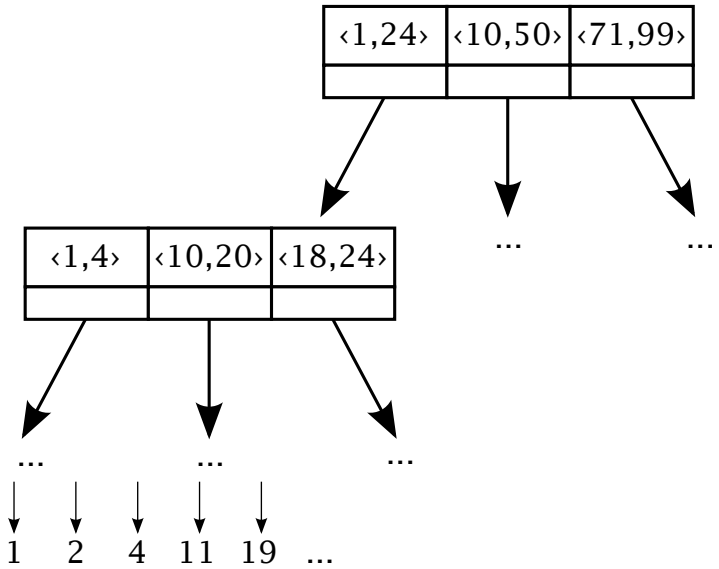
- **Eingabe:** Knoten L
- **Danach:** Alle Keys sind aktuell
- **Algorithmus:**
  1. Berechne $p =$ Union(L)
  2. Falls L Wurzel oder p in parent(L) aktuell: Abbrechen
  3. Rufe AdjustKeys(parent(L)) rekursiv auf

Gliederung

1. Motivation
2. Struktur eines GiST
3. Suche
4. Einfügen
5. **Effizienz**
6. Zusammenfassung

Effizienz von `Search(R, q)`

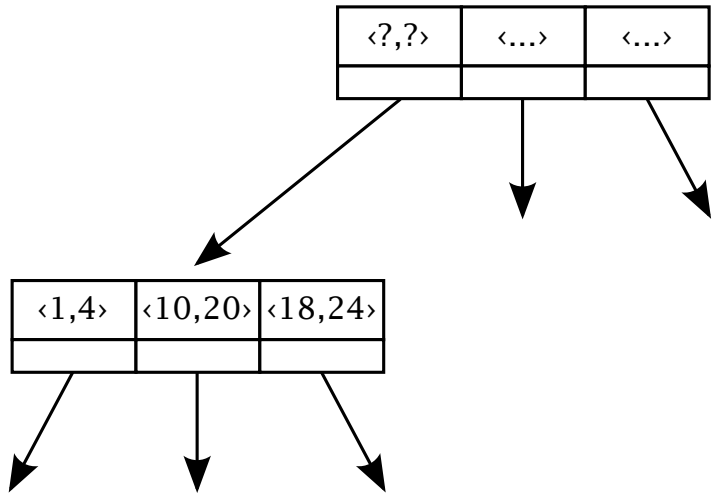
- B^+-Baum: $\mathcal{O}(\log n)$ (balancierter Suchbaum)
- Laufzeit: Anzahl paralleler Suchzweige entscheidend
  - Echte Überschneidung der Schlüssel
  - Genauigkeit der Heuristik für `Consistent`
  - Informationsverlust bei `Compress`, `Decompress`



- `Consistent` ist eine Heuristik für *SATISFIABILITY* (false positives erlaubt)
- *SATISFIABILITY* ist $\mathcal{NP}$-vollständig

Implementierung von `Insert(`R`, `q`)`

- Je genauer `Union` und `PickSplit`, desto weniger Überschneidungen
- Platzkomplexität: Mit `Compress`, `Decompress` nicht schlechter als im B^+-Baum

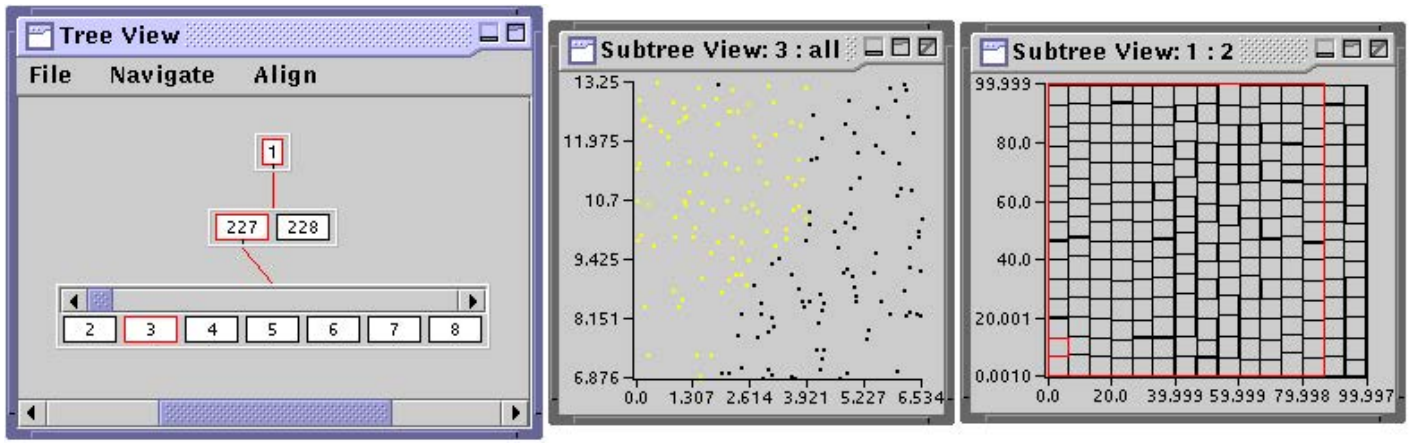


Gliederung

1. Motivation
2. Struktur eines GiST
3. Suche
4. Einfügen
5. Effizienz
6. **Zusammenfassung**

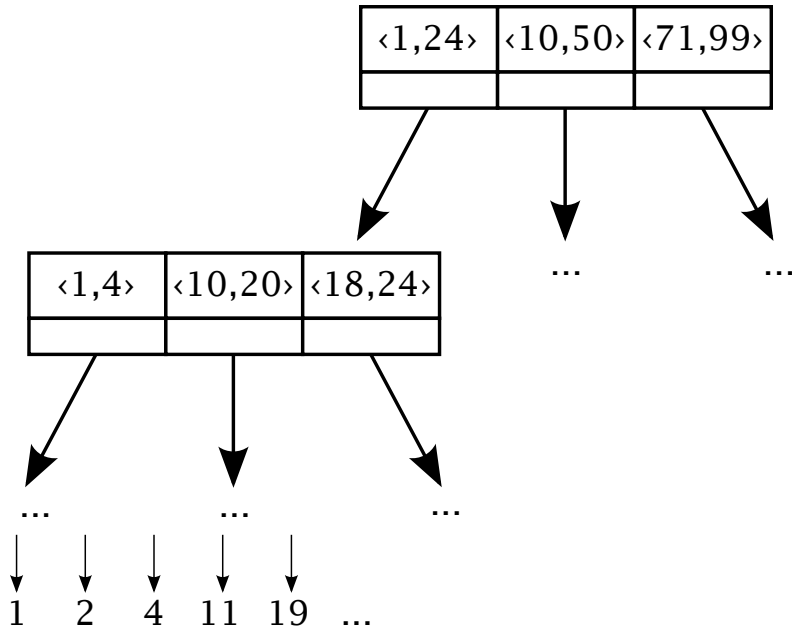
Zusammenfassung und Ausblick

- Implementation Issues [HNP95b]
  - Bulk Loading
  - Query Optimizer und Kostenschätzung
- GiST erleichtert Implementierung neuer Indexstrukturen
- Nur für Index-Bäume geeignet
- `libgist, amdb`: GiST-Implementierung und grafischer Debugger
- GiST-Implementierung für PostgreSQL [SB02]



Zusammenfassung

- Suche
  - `Search(`R`, `q`)`
  - `Consistent(`P`, `q`)`
- Einfügen
  - `Insert(`E`)`
  - `ChooseSubtree(`R`, `E`)`
  - `Penalty(`F`, `E`)`
  - `Split(`L`)`
  - `PickSplit(`P`)`
  - `Union(`P`)`
  - `AdjustKeys(`L`)`



Beispielimplementierung für Consistent

Konsistenz von Query- und Key-Prädikat

```
1
2 def consistent(k, q):
3   if q.type == 'equal':
4     return q.value in range(k.a, k.b)
5   elif q.type == 'contains':
6     return k.a <= q.b and k.b >= q.a
7   elif q.type == 'multiple':
8     return len([x for x in range(k.a, k.b + 1) \
9           if x%q.value == 0]) > 0
```

Quellen

HNP95a Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer: Generalized Search Trees for Database Systems. VLDB 1995: 562-573

HNP95b Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer: Generalized Search Trees for Database Systems. Technical Report #1274, University of Wisconsin at Madison, 1995.

SB02 Teodor Sigaev, Oleg Bartunov: GiST for PostgreSQL. `http://www.sai.msu.su/~megera/postgres/gist/` Aufgerufen am 03.06.2012